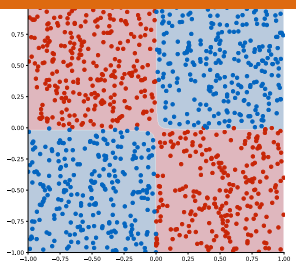
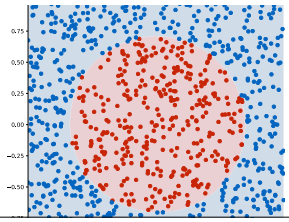
Linear Models 2:
Learning Non-linear Functions

Machine Learning 2019
mlvu.github.io

making linear models more powerful

d	p	d * p
0.75	0.98	0.74
-0.66	-0.32	0.21
-0.45	0.84	-0.38
0.93	0.78	0.72
-0.42	0.24	-0.10
-0.02	0.43	-0.01
-0.74	0.58	-0.43
-0.41	-0.41	0.17
0.59	0.72	0.42

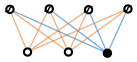




2

A few lectures ago, we saw how we could make a linear model more powerful, and able to learn nonlinear decision boundaries by just expanding our features: we add combinations of existing features, and depending on which combinations we add, we can learn new, non-linear decision boundaries or regression surfaces.

from linear to nonlinear models

neural networks specifically the feedforward network	SVMs using the kernel trick	
$\sigma(\mathbf{w}^T\mathbf{x} + b)$	$\mathbf{w}^T\mathbf{x} + b$	linear model
	$k(\mathbf{x}^i, \mathbf{x}^j)$	feature extractor
learns a feature extractor together with the classifier	uses a kernel to expand the feature space	

3

Both models we will see today (neural networks and support vector machines) take this idea and build on it. Neural networks are a big family, but the simplest type, the two-layer feedforward network functions as a feature extractor followed by a linear model. In this case, we don't choose the extended features but we *learn* them, together with the weights of the linear model.

The SVM doesn't learn the expanded features (we still have to choose them manually), but it uses a *kernel function* to allow us to fit a linear model in a very high-dimensional feature space very cheaply.

linear models 2

part 1: 1985–1995
(feedforward) Neural Networks

Backpropagation

part 2: 1995–2005
Support Vector Machines, Hinge loss

The Kernel Trick

4

The layout of today's lecture will be largely chronological. We will focus on neural networks, which were very popular in the late eighties and early nineties. Then, towards the end of the nineties, interest in neural networks died down a little and SVMs became much more popular (we'll discuss why).

In the next lecture, we'll focus on Deep Learning, which sees neural networks come back in a big way.

---

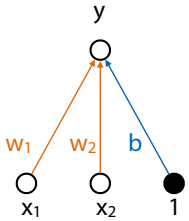
neuron

In the very early days of AI (the late 1950s), researchers decided to take a simple approach to AI. The brain is the only truly intelligent system we know, so let's see what it's made of, and if that provides some inspiration for intelligent (and learning) computer systems.

They started with a single brain cell: a neuron. A neuron receives multiple different signals from other cells through connections called **dendrites**. It processes these in a relatively simple way, deriving a single new signal, which it sends out through its single **axon**. The axon branches out so that the single signal can reach other cells.

image source: ***http://www.sciencealert.com/scientists-build-an-artificial-neuron-that-fully-mimics-a-human-brain-cell***

1957: the perceptron



$y = w_1x_1 + w_2x_2 + b$

output `man` if $y > 0$
output `woman` otherwise

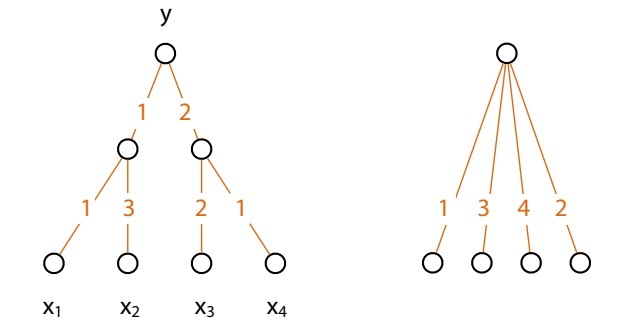
This principles needed to be radically simplified to work with computers of that age, but doing so yielded one of the first successful machine learning systems: the perceptron (also seen in the **video** in the first lecture).

The perceptron had a number of inputs (the features in modern parlance), each of which was multiplied by a weight. These result was summed, together with a bias parameter, and the sign of this result was used as the classification.

Of course, we've seen this classifier already: it's just our basic linear classifier. The training algorithm was a little different from gradient descent, but the basic principle was the same.

Note that the intercept can be represented as just another input that we just fix to always be 1. This is called a **bias node**.

problem: composing neurons



$y = 1(1x_1 + 3x_2) + 2(2x_3 + 1x_4)$ $y = 1x_1 + 3x_2 + 4x_3 + 2x_4$

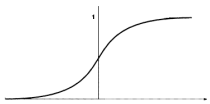
7

Of course the brain's power does not come from the power of a single neuron: it's the composition of many simple parts that allows it to do what it does. And this is where the perceptron turns out to be too simple an abstraction. Because composing perceptrons (making the output of one perceptron the input of another) doesn't make it more powerful. All you end out with is something that is equivalent to another linear model. We're not creating models that can learning non-linear functions.

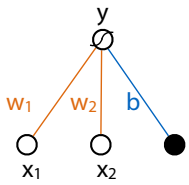
We've removed the bias node here for clarity, but that doesn't affect our conclusions: any composition of affine functions is itself an affine function.

nonlinearity

sigmoid

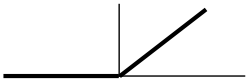


$$\sigma(x) = \frac{1}{1+e^{-x}}$$



$$y = \sigma(w_1x_1 + w_2x_2 + b)$$

ReLU



$$r(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

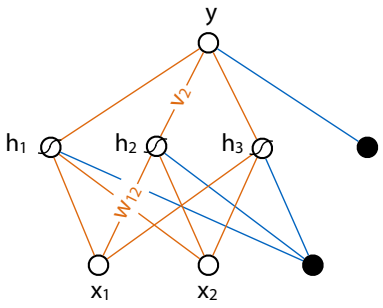
8

The simplest solution is to apply a **nonlinear function** to each neuron, called th**e activation function,** after all the weighted inputs have been combined. One popular option (especially in the early days) is the logistic sigmoid, which we've seen already. Applying a sigmoid means that the sum of the inputs can range from negative infinity to positive infinity, but the output is always in the interval [0, 1].

Another, more recent nonlinearity is the linear rectifier, or **ReLU** nonlinearity. This function just sets every negative input to zero, and keeps everything else the same.

Not using an activation function is also called using a **linear activation**.

feedforward network



output layer

hidden layer

input layer (features)

aka Multilayer Perceptron (MLP)

9

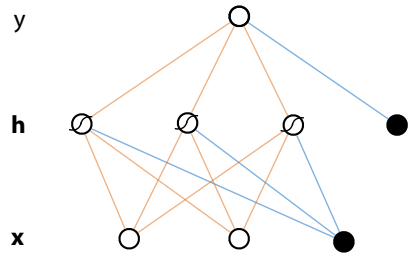
Using these nonlinearities, we can arrange single neutrons into **neural networks**. Any arrangement makes a neural network, but for ease of training, this arrangement was the most popular for a long time. It's called the **feedforward network** or **multilayer perceptron**. We arrange a layer of hidden units in the middle, each of which acts as a perceptron with a nonlinearity, connecting to all input nodes. Then we have one or more output nodes, connecting to all hidden layers. Crucially:

- There are no cycles, the network feeds forward from input to output.
- Nodes in the same layer are not connected to each other, or to any other layer that the previous one.
- Each layer is **fully connected** to the previous layer, every node in one layer connects to every node in the layer before it.

In the 80s and 90s they usually had just one hidden layer, because we hadn't figured out how to train deeper networks.

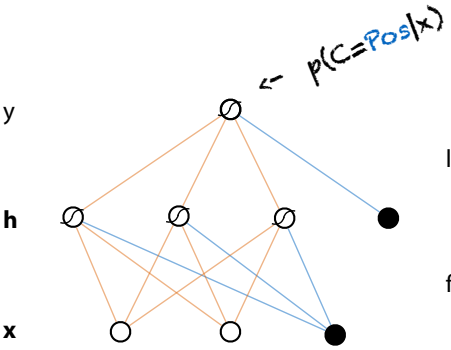
Every orange and blue line in this picture represents one

regression



To build a regression model, all we need is one output node without an activation. This means that our network describes a function from the feature space to the real number line.

classification (binary)



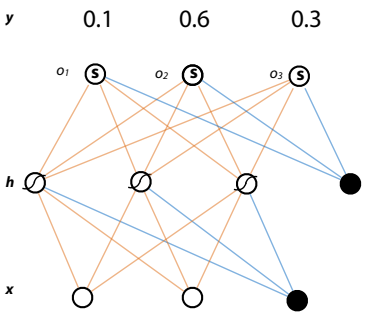
For classification, we can do what the perceptron did (use the sign of the output as the class), but it's more common these days to make the output probabilistic.

Just like we did in logistic regression, we apply the logistic sigmoid to the output and interpret the resulting value as the probability that the given input (x) is of the positive class.

We can then train using cross-entropy loss (more on that later).

classification (multiclass)

softmax activation



$$o_i = \mathbf{w}^T \mathbf{h} + b$$

$$y_i = \frac{\exp(o_i)}{\sum_j \exp(o_j)}$$

For multi class classification, we can use something called a **softmax activation**. We create a single output node per class, and ensure that they sum to one.

Softmax simply takes the exponent of each output node (to ensure that they are all positive) and then divides each by the total (to ensure that they sum to one).

After the softmax we can interpret the output of node y_3 as the probability that **x** has class 3, and train with cross entropy loss.

stochastic gradient descent

pick random weights **w** (for the whole model)

loop:

  for x in X:

    $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla \text{loss}_{\mathbf{x}}(\mathbf{w})$

13

Because a neural networks can be to compute (considering we need to do many steps of gradient descent) we tend to use **stochastic gradient descent** to train them.

Stochastic gradient descent is very similar to the gradient descent we've seen already, but we define the loss function over a single example instead of over the whole dataset (just use the same loss function, but pretend your data set consists of only one instance. Stochastic gradient descent has many advantages, including:

- Using a new instance each time adds some randomness to the process, which can help to escape local minima.
- Gradient descent works fine if the gradient is not perfect, but good on average (over many steps). This means that taking many small inaccurate steps is much better than taking one very accurate big step.
- Computing the loss over the whole data is expensive. By computing loss over one instance, we get N steps of stichastic gradient descent for the price of one step of regular gradient descent.

---

summary: training a neural network

get some examples of input and output

get a loss function
least squares, cross entropy

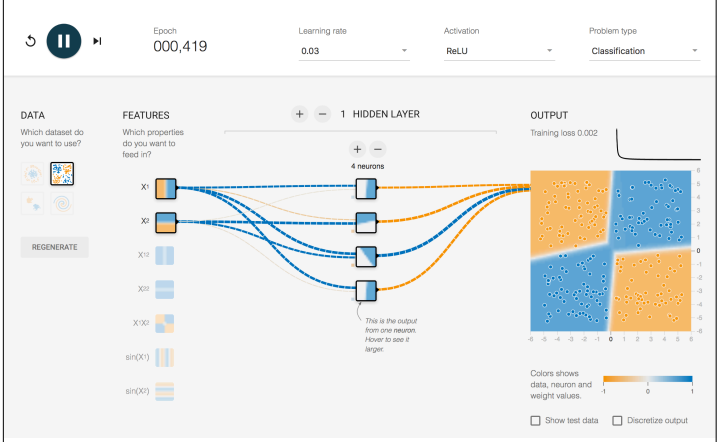
work out the gradient of the loss wrt the weights

use (stochastic) gradient descent to improve the weights bit by bit.

14

---

playground.tensorflow.org



15

Before we dig into the details, we can get a sense of what this looks like in tensorflow playground. Note:

- How the shape of the decision boundary changes based on the activation functions we choose (curvy for sigmoid, piecewise linear for ReLU)
- That adding another layer makes the network much more difficult to train (especially with sigmoid activations).

But how do we compute the gradient for such complex models?

16

options

Symbolically: too expensive

Numerically: too unstable, also expensive

Middle ground: **backpropagation**

17

If we describe our model as a composition of modules, the gradient is the product of the gradient of each module with respect to its arguments.

18

This is the basic principle behind backpropagation: if we have a function, that is a composition of other functions, we can write out the derivative as repeated applications of the chain rule.

1974, 1980's: backpropagation

Break your computation down into a chain of *modules*.

Work out the derivative of each module with respect to its input symbolically.

Compute the global gradient *for a given input* by multiplying these gradients.

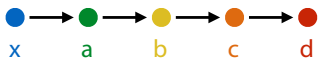
example

$$f(x) = \frac{2}{\sin(e^{-x})} \qquad f(x) = d(c(b(a(x))))$$

modules:

$$d(c) = \frac{2}{c}$$
$$c(b) = \sin b$$
$$b(a) = e^{a}$$
$$a(x) = -x$$

computation graph:



To show that backpropagation is a generic algorithm for working out gradients, we'll first show how it works for some arbitrary function. Defining the sub functions a, b, c, and d as shown, we can write f(x) = d(c(b(a(x)))).

The graph on the right is a **computation graph**: each node represents a small computer program that receives an input, computes an output and passes it on to another module.

Normally, we wouldn't break a function up in such small modules. This is just a simple example to illustrate the principle.

chain rule

$$\frac{\partial f(x)}{\partial x} = \frac{\partial d(c(b(a(x))))}{\partial c(b(a(x)))} \frac{\partial c(b(a(x)))}{\partial x} = \frac{\partial d}{\partial c} \frac{\partial c}{\partial x}$$

$$\frac{\partial d}{\partial x} = \frac{\partial d}{\partial c} \frac{\partial c}{\partial b} \frac{\partial b}{\partial a} \frac{\partial a}{\partial x}$$

Because we've described our function as a composition of modules, we can work out the derivative purely by repeatedly applying the chain rule.

Since we know for each function what the argument is, we'll leave the arguments out to keep the notation clean.

example

$$f(x) = \frac{2}{\sin(e^{-x})}$$

$$d(c) = \frac{2}{c}$$
$$c(b) = \sin b$$
$$b(a) = e^{a}$$
$$a(x) = -x$$

$$\boxed{\frac{\partial f}{\partial x}} = \boxed{\frac{\partial d}{\partial c}}\frac{\partial c}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{\partial x}$$

global derivative

local derivatives

22

We'll call the derivative of the whole function the **global derivative**, and the derivative of each module with respect to its input we will call a **local derivative**.

---

backpropagation

Write your function as a composition of **modules**.
What the modules are is up to you.

Work out the local derivative of each module **symbolically**.

Do a **forward pass** for a given input x.
I.e. compute the function f(x), remember the intermediate values

Compute the local derivatives for x, and multiply to compute the global derivative.

23

More fine-grained modules make the local derivatives easier to work out, but increase numeric instability.

---

work out *local* derivatives

symbolically

$$d(c) = \frac{2}{c}$$
$$c(b) = \sin b$$
$$b(a) = e^{a}$$
$$a(x) = -x$$

$$\frac{\partial f}{\partial x} = \frac{\partial d}{\partial c}\frac{\partial c}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{\partial x}$$

$$\frac{\partial f}{\partial x} = -\frac{2}{c^2} \cdot \cos b \cdot e^{a} \cdot -1$$

24

compute a forward pass (x=-4.499)

retain values of a, b, c, d

$$f(-4.499) = 2$$

$$d = \frac{2}{c} = 2$$

$$c = \sin b = 1$$

$$b = e^{a} = 90$$

$$a = -x = 4.499$$

25

---

compute the backward pass

numerically

$$f(-4.499) = 2$$

$$d = \frac{2}{c} = 2$$

$$c = \sin b = 1$$

$$b = e^{a} = 90$$

$$a = -x = 4.499$$

$$\frac{\partial f}{\partial x} = -\frac{2}{c^2} \cdot \cos b \cdot e^{a} \cdot -1$$

$$= -\frac{2}{2^2} \cdot \cos 90 \cdot e^{4.499} \cdot -1$$

$$= -\frac{1}{2} \cdot 0 \cdot 90 \cdot -1 = 0$$

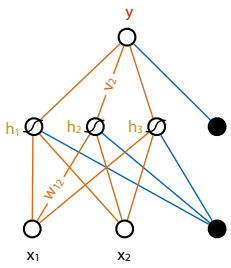
26

Now we just fill in the intermediate values for the local derivatives, and compute the product numerically.

---

backprop for a feedforward network

t: training data



$$\frac{\partial l}{\partial v_2}, \frac{\partial l}{\partial w_{12}}$$

$$l = (y - t)^2$$

$$y = v_1 h_1 + v_2 h_2 + v_3 h_3 + b_y$$

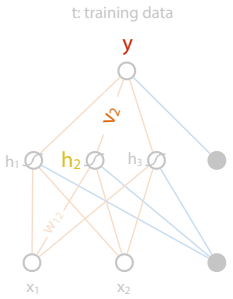
$$h_2 = \frac{1}{1 + \exp(-h_2')}$$

$$h_2' = w_{12} x_1 + w_{22} x_2 + b_2$$

27

Let's see how this works for a neural net. Remember that we don't care about the derivative of the output with respect to x. We want to know **the derivative of the loss with respect to the weights**. To keep the notation clear, we'll leave out the arguments of functions.

NB: We're slightly deviating from the notation in the second lecture: **y** is the output of the model, **t** is the target label/value.



$$l = (y - t)^2$$

$$y = v_1 h_1 + v_2 h_2 + v_3 h_3 + b_h$$

$$h_2 = \frac{1}{1 + \exp(-h_2')}$$

$$h_2' = w_{21} x_1 + w_{22} x_2 + b_x$$

$$\frac{\partial l}{\partial v_2} = \frac{\partial l}{\partial y} \frac{\partial y}{\partial v_2}$$

$$= 2(y - t) \cdot h_2$$

$$v_2 \leftarrow v_2 - \eta \cdot 2(y - t) \cdot h_2$$

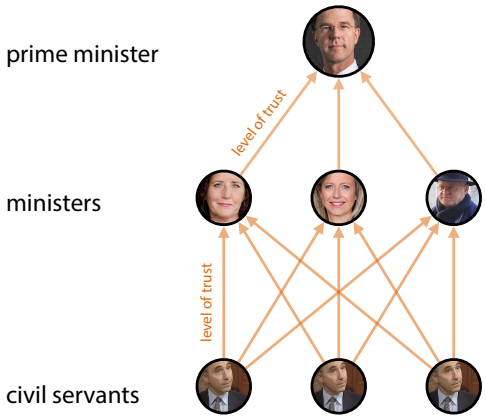
28

Here's what the local gradients look like for the weight v_2.

The line on the bottom shows how we update v_2 when we apply a single step of stochastic gradient descent for x (x may not appear in the gradient, but the values y and h2 we computed using x).

---

backpropagation by analogy



29

To see what this update rule means, we can use an analogy. Think of the neural network as a hierarchical structure like a government trying to make a decision. The output node is the prime minister: he provides the final decision (for instance what the tax on cigarettes should be).

To make this decision, he listens to his ministers. His ministers don't tell him what to do, they just shout. The louder they shout, the higher they want him to make the output.

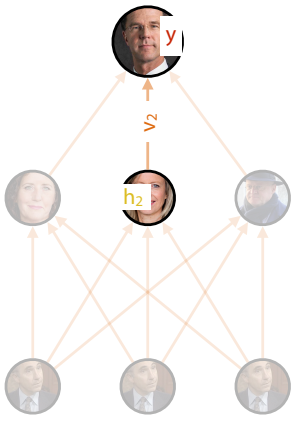
If he trusts a particular minister, he will weigh their advice positively, and follow their advice. If he distrusts the minister, he will do the opposite of what the minister says. The ministers each listen to a bank of civil servants and weigh their opinions in the same way the prime minister weight theirs. All ministers listen to the same civil servants, but they have their own level of trust for each.

(We haven't drawn the bias, but you can think of the bias as the prime minister's own opinion; how strong the opinions of the ministers need to be to change his mind).

image sources:

---

backpropagation by analogy



$$v_2 \leftarrow v_2 - \eta \cdot 2(y - t) \cdot h_2$$

30

So let's say the network has produced an output. The prime minister has set a tax on cigarettes y, and based on the consequences realises that he should have set a tax of t. He's now going to adjust his level of trust in each of his subordinates.

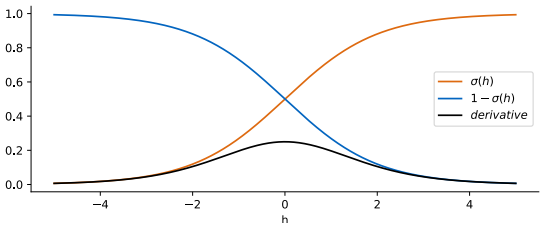
Looking at the update rule for weight v_2, we can see that he takes two things into account: the error (y-t), how wrong he was, and what minister h_2 told him to do.

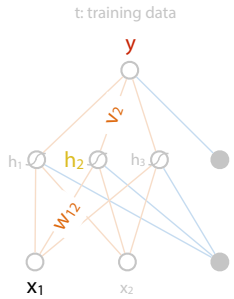
- If the error is positive, he set y too high. If h_2 shouted loudly, he will lower his trust in her.
- If the error is negative, he set y too low. If h_2 shouted loudly, he will increase his trust in her.

If we use a sigmoid activation, the ministers can only provide values between 0 and 1. If we use an activation that allows h2 to be negative, we see that the minister takes the sign into account: if h2 was negative and the error was negative too, the trust in the minister increases (because the PM should've listened to her).

reminder: derivative of the sigmoid

$$\frac{\partial \sigma(h)}{\partial h} = \sigma(h)(1 - \sigma(h))$$





$$l = (y - t)^2$$
$$y = v_1 h_1 + v_2 h_2 + v_3 h_3 + b_h$$
$$h_2 = \frac{1}{1 + \exp(-h_2')}$$
$$h_2' = w_{21} x_1 + w_{22} x_2 + b_x$$

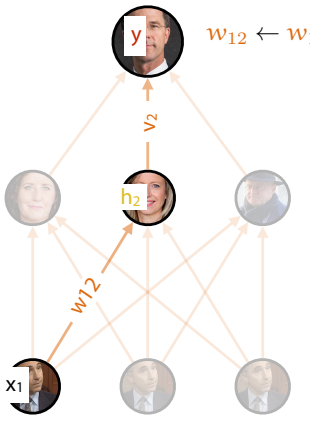
$$\frac{\partial l}{\partial w_{12}} = \frac{\partial l}{\partial y} \frac{\partial y}{\partial h_2} \frac{\partial h_2}{\partial h_2'} \frac{\partial h_2'}{\partial w_{12}}$$

$$= 2(y - t) \cdot v_2 \cdot h_2(1 - h_2) \cdot x_1$$

$$w_{12} \leftarrow w_{12} - \eta \cdot 2(y - t) \cdot v_2 \cdot h_2(1 - h_2) \cdot x_1$$

So far, this is no different from gradient descent on a linear model. The real power of the backpropagation algorithm shows when we look at how the error propagates back down the network (hence the name) and is used to update the weights. Lets look at the derivative for weight w12

backpropagation by analogy



$$w_{12} \leftarrow w_{12} - \eta \cdot 2(y - t) \cdot v_2 \cdot h_2(1 - h_2) \cdot x_1$$

global error

h_2's error

h_2''s contribution, considering the activation function

w_{12}'s contribution

summary

Neural networks are networks of neurons: linear units with nonlinear activations.

The simplest version is the feedforward network.

Trained by stochastic gradient descent.

To avoid working out the entire gradient, we use backpropagation: work out local derivatives, and fill in the values from the forward pass.

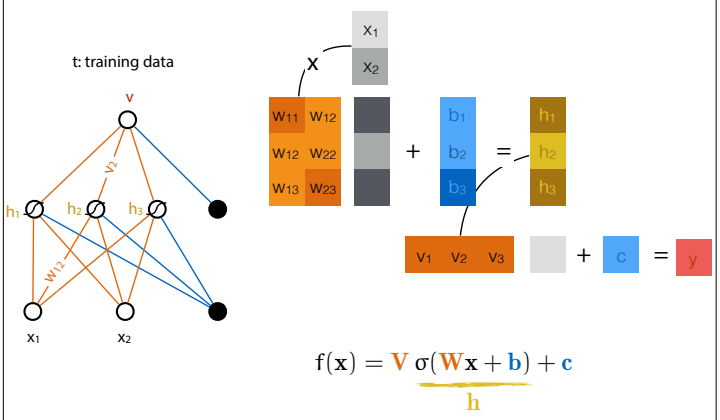
'90–'95: the start of the neural network winter

Why did interest in Neural Networks die out?

- NNs are non-convex, difficult to train.
  SVMs have convex loss, optimal solution guaranteed
- A single forward backward pass was very expensive.
  Many passes required to train an NN
- They're not that great as classifiers/regression models
  They're not bad, but their power is in their versatility (more next week).
- We didn't have good libraries/abstractions. Too many possibilities.

These weren't just reasons not to use neural nets in production. They also slowed down the research on neural nets. SVM researchers were (probably) able to more faster, because once they'd designed a kernel, they could compute the optimum performance and know, without ambiguity, whether it worked or not. Neural net researchers could design an architecture and spend months tuning the training algorithm without ever knowing whether the architecture would eventually perform.

next week: it's all just linear algebra



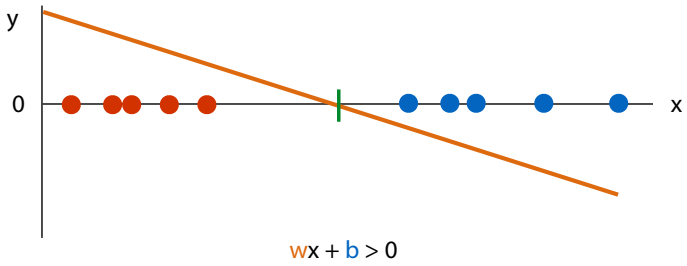
$$f(\mathbf{x}) = \mathbf{V}\,\sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) + \mathbf{c}$$

$\mathbf{h}$

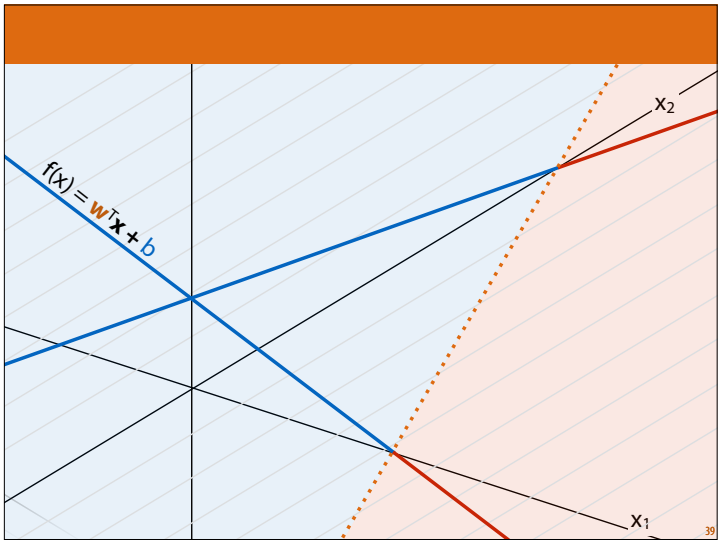
One important part of building such a framework is to recognise that all of this can easily be described as matrix multiplication/addition, together with the occasional element-wise non-linear operation. This allows us to write down the operation of a neural network very elegantly. In order to make proper use of this, we should also work out how to do the backpropagation part in terms of matrix multiplications. That's where we'll pick up next week in the first **deep learning** lecture.

break

1D linear classifier

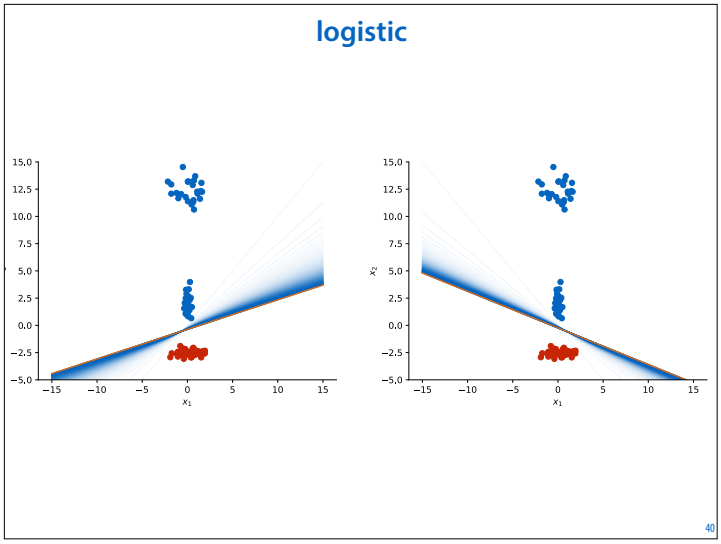


38

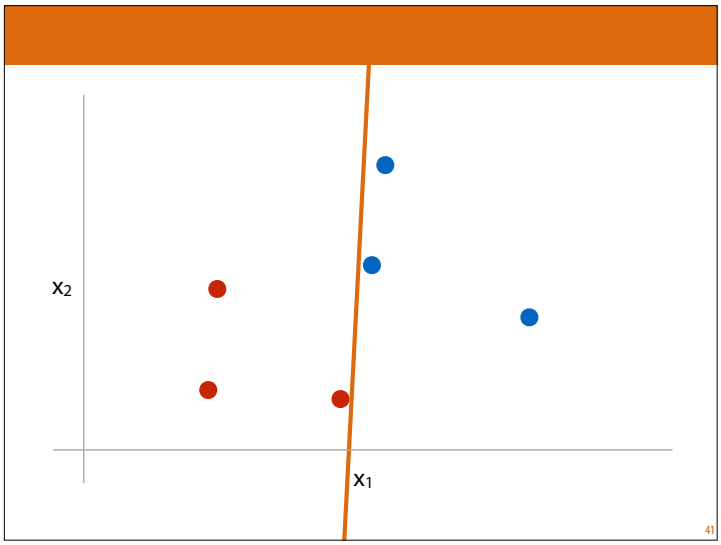


39

The function f(x) = $\mathbf{w}^T\mathbf{x}$ + b describes a linear function from our feature space to a single value. Here it is in 2D: a plane that intersects the feature space. The line of intersection is our decision boundary.



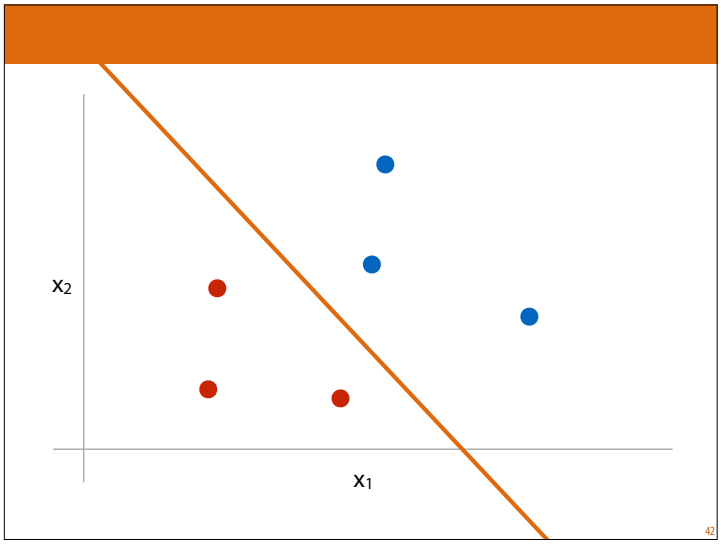
This is where we left things last time. We saw that the cross-entropy loss performed very well, but it had one problem: when the data are very well separable, it didn't have any basis to choose between two models like this. Both separate the training data very well. Yet, they're very different models.



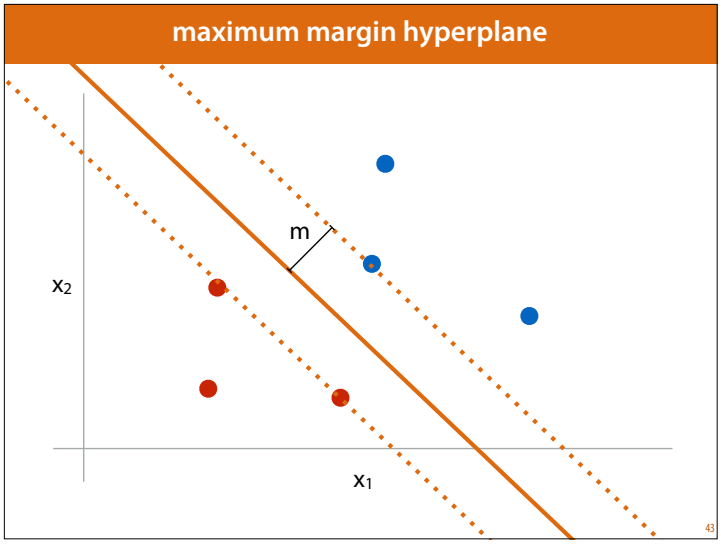
Here is an extreme example.This decision boundary separates the data perfectly. And yet, if I see a new instance that is very similar to the rightmost red point, but with a slightly higher x_1 value, it suddenly becomes a blue point.

This illustrates the intuition behind our final loss function

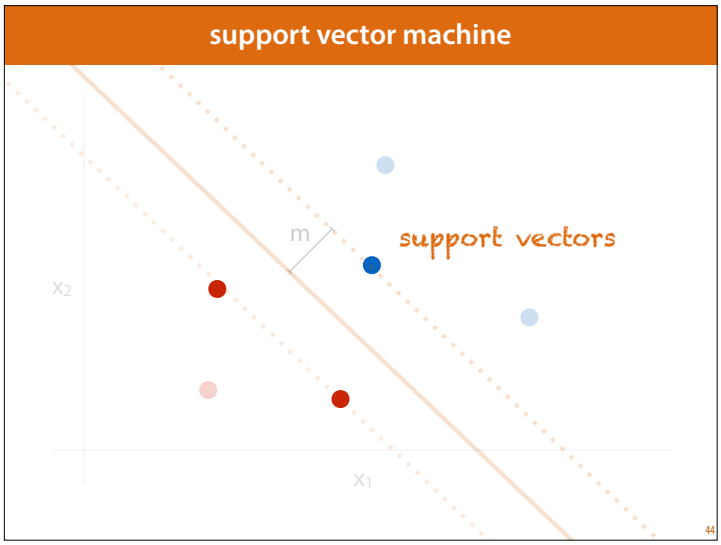
---



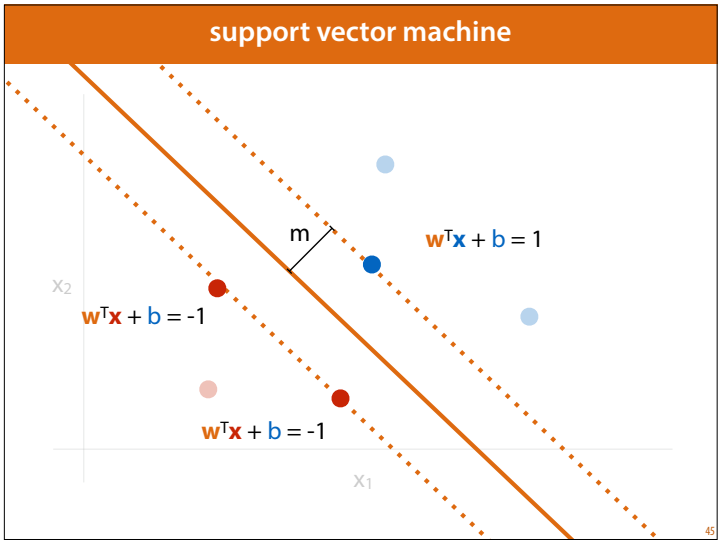
What we are looking for is the hyperplane that has a maximal distance to the nearest positive and nearest negative point.



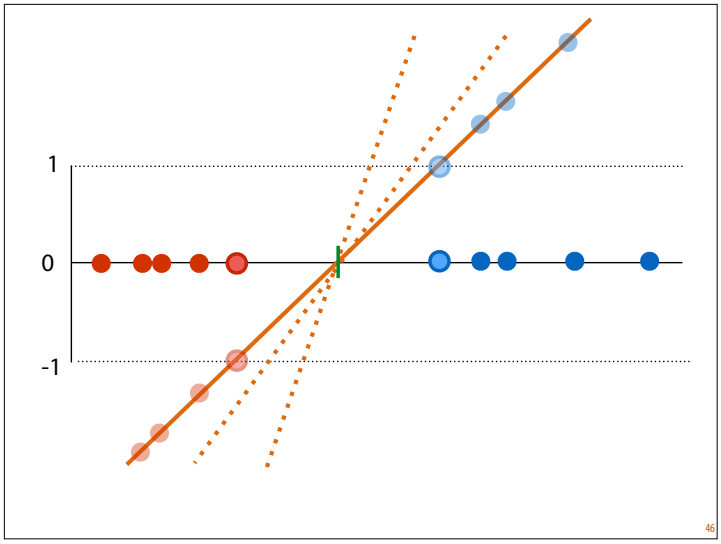
We call this distance the **margin** and this is what we want to maximize.



The points closest to the decision boundary are called the **support vectors** (because they can be used to describe the decision boundary). The distance to the support vectors is called the **margin**. We'll assume that the decision boundary is chosen so that the margin is the same on both sides.



Remember that there are essentially infinitely many hyperplanes that define this decision boundary. Because of this we can decide that the hyperplane we use is always the one that crosses the negative support vectors at -1, and the positive support vectors at 1.



Here's what that looks like for a 1D feature space. Any of the lines drawn determine the same decision boundary. We just decide to choose the one that crosses the support vectors at -1 and 1 (the solid line).

But how do we choose the support vectors? (It's easy here, but in higher dimensions, it's not so obvious).

---

objective

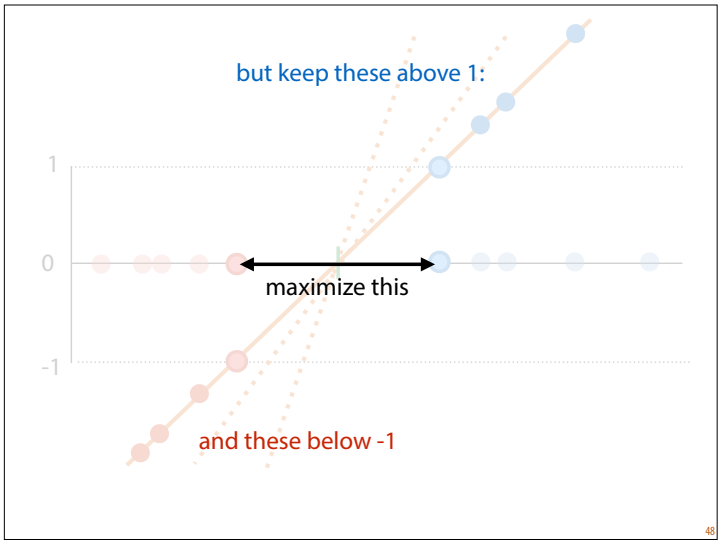
maximize "2x the size of the margin"

such that :

$$\mathbf{w}^T\mathbf{x}^i + b \geqslant 1 \quad \text{for } \mathbf{x}^i \in X^P$$
$$\mathbf{w}^T\mathbf{x}^i + b \leqslant -1 \quad \text{for } \mathbf{x}^i \in X^N$$

47

We can choose the support vectors by using a **constrained** optimisation objective. As before, we state a function that we want to minimise or maximise, but this time, we also add some constraints.



objective

maximize "2x the size of the margin"

such that :

$$\mathbf{w}^T\mathbf{x}^i + b \geqslant 1 \quad \text{for } \mathbf{x}^i \in X^P$$
$$\mathbf{w}^T\mathbf{x}^i + b \leqslant -1 \quad \text{for } \mathbf{x}^i \in X^N$$

such that :

$$y^i(\mathbf{w}^T\mathbf{x}^i + b) \geqslant 1 \quad \text{for all } \mathbf{x}^i$$

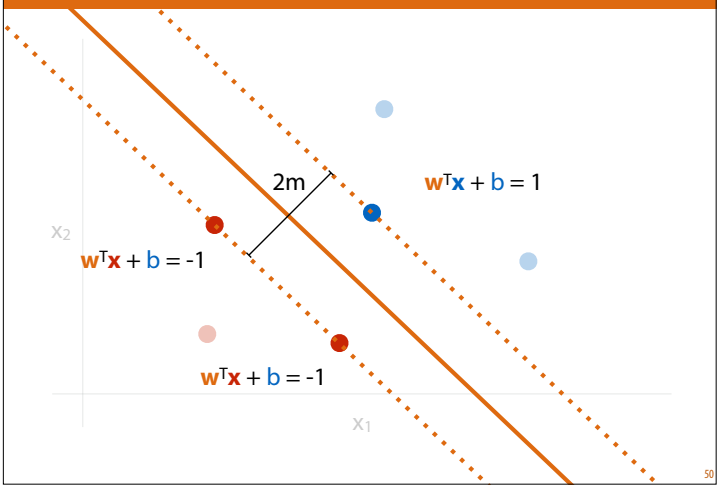
49

We can simplify the constraints by introducing a label y^i for each point xi which is -1 for negative points and +1 for positive points.

(Similar to the label we introduced for the least squares loss).

---

what is the size of the margin?

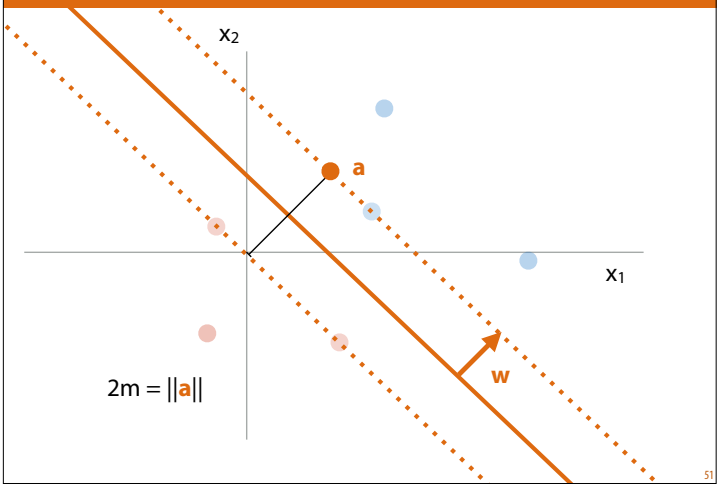


50

The odd thing is that the support vectors change as we changes the parameters of the hyperplane, but that doesn't lead to discontinuities in our loss landscape (as it did with the accuracy. The reason is we can compute the value "2x the size of the margin" *without knowing what the support vectors are.*

---

what is the size of the margin?



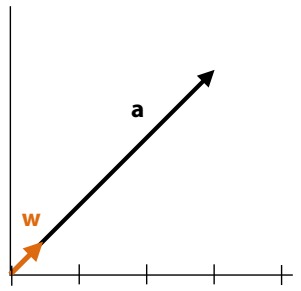
51

To make the math easier, let's move the axes around so that the lower dotted line (belonging to the negative support vectors) crosses the origin. Doing this doesn't change the size of the margin.We can now imagine vector from the origin to the upper dotted line, which we'll call **a**. The length of this vector is exactly the quantity we're interested in.

what is the size of the margin?

$$w^T 0 + b = -1$$
$$w^T a + b = 1$$

$$w^T a = 2$$
$$||w||\,||a|| = 2$$
$$||a|| = \frac{2}{||w||}$$



$$w^T x = ||w||\,||x|| \cos\alpha$$

52

objective

maximize : $\frac{2}{||w||}$

such that :

$$y^i(w^T x^i + b) \geqslant 1 \quad \text{for all } x^i$$

53

Note that almost all the complexity of the loss is in the constraints. Without them we could just let all elements of **w** go to zero

However, the objective function requires the output of our model to be larger than 1 for all positive points and smaller than -1 for all negative points. This will automatically push the margin up to the support vectors (if there we room between the margin and the nearest points, we could increase the margin without violating the constraints)

objective: hard margin SVM

minimize : $\frac{1}{2}||w||$

such that :

$$y^i(w^T x^i + b) \geqslant 1 \quad \text{for all } x^i$$

54

We can now write our objective function in a more mathematical way. This is called a "hard margin" SVM, since no points are allowed to violate the margin.

We take the inverse of the earlier objective, because we prefer to minimise (that is, we want a *loss* function).

The hard margin classifier is nice, but it doesn't work well when:

- We have data that is not linearly separable
- We have a very nice decision boundary if we just ignore a few misclassified points (i.e. a little noise).

soft margin SVM

minimize : $\frac{1}{2}||w|| + C\sum_i p_i$

such that :

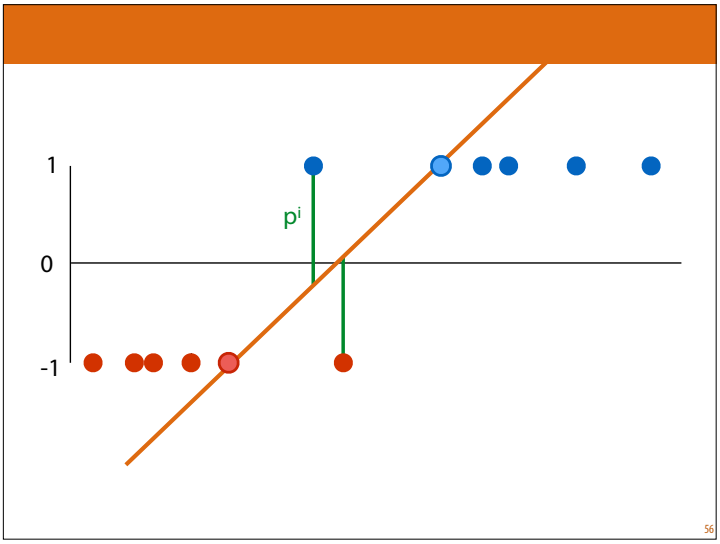
$$y^i(w^T x^i + b) \geqslant 1 - p^i \quad \text{for all} \quad x^i$$

$$p^i \geqslant 0$$

55

To deal with this, we can use a soft margin SVM. p_i is a **slack parameter.** It relaxes the constraint a little, to allow *some* points to fall in the margin, but we pay a price in the loss function (which gets higher if we allow bigger penalties). Any point for which pi is not zero is on the wrong side of the decision boundary.

C is a hyperparameter, indicating how much we care if the margins are violated.

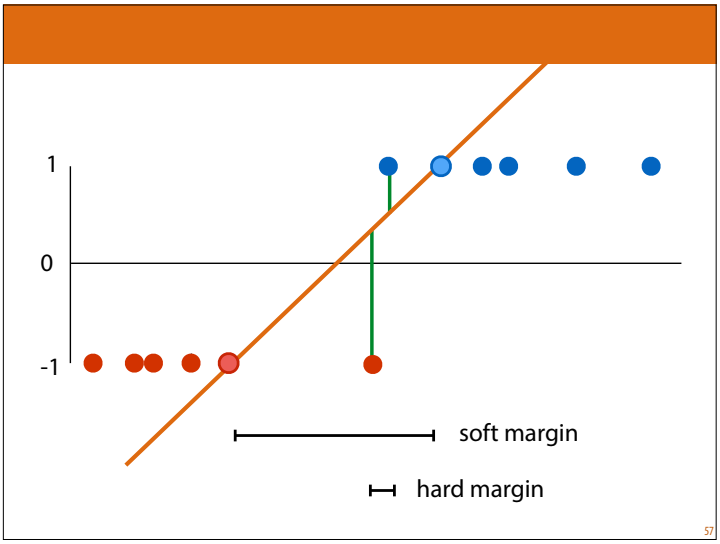


56

Here is what that looks like in 1D. The open points are the support vectors, and for each class, we have one point on the wrong side of the decision boundary (requireing us to pay the residual p^i as a penalty.

So, the objective function has a penalty added to it, making it bigger. But, the margin we are now able to create is much wider. If we used a hard margin classifier,

C lets us trade off these objectives.



57

Even if the points are linearly separable, it can be good to allow a little slack. Here, the two points that would be the support vectors of the hard margin objective leave a very narrow margin. By allowing a little slack, we can get a much wider margin that provides a decision boundary that is more likely top generalise to unseen data (because it is more evenly between the two point clouds.

a fork in the road

option one: express everything in terms of **w**, get rid of the constraints.

- Allows gradient descent to be used.
- Good for use with neural networks/deep learning.

option two: express everything in terms of the support vectors, get rid of **w**.

- Doesn't allow error to propagate back, but …
- allows the **kernel trick** to be applied.

option one

$$p^i = \max\left(0, y^i(w^T x^i + b) - 1\right)$$

minimize :

$$\frac{1}{2}||w|| + C\sum_i \max\left(0, y^i(w^T x^i + b) - 1\right)$$

~~such that :~~

We rewrite the constraints in terms of p^i, and insert these terms into the loss function.

This gives us a loss function we can directly apply to any model. For instance when training a neural network to classify, this makes a solid alternative to cross-entropy loss. This is sometimes called the L1-SVM (loss). There is also the L2-SVM loss where a square is applied to the p^i function, to increase the weight of outliers.

classification losses

~~Least squares loss (today)~~

~~Log loss / logistic regression (week 3, Probability 1)~~

~~SVM loss (week 3, Linear Models 2)~~

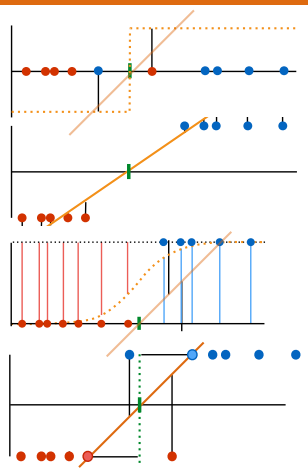
loss functions

accuracy
aka zero-one loss, nr. of misclassified instances
doesn't work with gradient descent

least squares
assign -1, 1 to points treat as regression
doesn't really work well

cross-entropy
use a sigmoid to turn the linear output into probabilities
works well for non-non-separable data, overfits, has multiple solutions

soft margin SVM
aka Hinge loss, maximum margin loss
works well in high-dim data, separable data



61

option two: kernel SVM

$$\text{minimize}: \frac{1}{2}||\mathbf{w}|| + C\sum_i p_i$$

such that :

$$y^i(\mathbf{w}^T\mathbf{x}^i + b) \geqslant 1 - p^i \quad \text{for all} \quad \mathbf{x}^i$$
$$p^i \geqslant 0$$

62

For option two, we're going to leave the constraints, but rewrite them. Instead, we will get rid of the parameters **w** and b. We will end up with a formulation of the decision boundary **purely in terms of the support vectors**. Moreover, it will never refer to a single instance x^i in isolation, only in terms of the dot product of pairs of instances.

optimising under constraints

$$\text{minimize } f(a) = a^2$$
$$\text{such that } a \geqslant 1$$

$$L(a, \alpha_1, \ldots, \alpha_n) = a^2 - \alpha(1 - a)$$

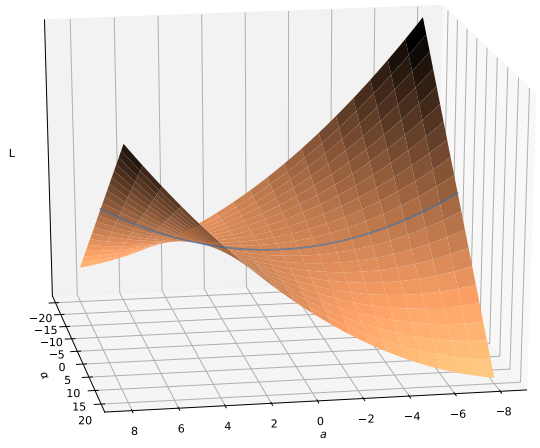
$$\text{solve } \nabla L = 0$$
$$\text{such that } \alpha \geqslant 0$$

63

To do this, we'll need to look at how we actually perform optimization under constraints.

The standard method is to use Lagrange multipliers. We don't have room for a full treatment, but we'll quickly illustrate the basic idea. We add the extent to which we violate each constraint to the loss function together with a weight alpha (which becomes an extra parameter of the function).

This new function L has an optimum where the original function is minimal within the constraints. The new optimum is a saddlepoint, so we have to set the gradient equal to zero. We can't solve by gradient descent. Note also that we haven't *removed* the constraint, we have merely formulated a *dual function*; another optimisation objective with the same solution.



64

The line is the original objective function. The surface indicates how far we've strayed from the objective.

Lagrange multipliers (under KKT conditions)

$$\begin{aligned} &\text{minimize } f(a) \\ &\text{such that } g_i(a) \geqslant 0 \quad \text{for } i \in [1, n] \end{aligned}$$

$$L(a, \alpha_1, \ldots, \alpha_n) = f(a) - \sum_i \alpha_i g_i(a)$$

$$\begin{aligned} &\text{solve } \nabla L = 0 \\ &\text{such that } \alpha_i \geqslant 0 \quad \text{for } i \in [1, n] \end{aligned}$$

65

The method of Lagrange multipliers gives us a new objective function to minimize, with additional parameters for the constraints.

$$\begin{aligned} &\text{minimize } \frac{1}{2} w^T w + C \sum_i p^i \\ &\text{such that } y^i (w^T x^i + b) \geqslant 1 - p^i \\ &\qquad p^i \geqslant 0 \end{aligned}$$

$$\begin{aligned} &\text{minimize } -\frac{1}{2} \sum_i \sum_j \alpha^i \alpha^j y^i y^j x^{i\,T} x^j + \sum_i \alpha^i \\ &\text{such that } 0 \leqslant \alpha^i \leqslant C \\ &\qquad \sum_i \alpha^i y^i = 0 \end{aligned}$$

66

If we do this for the SVMs (and do a lot of rewriting), we end up with this optimisation objective.

We'll skip the derivation this time around. Check last year's lecture if you're interested.

image source: https://www.flaticon.com/free-icons/road

$$\text{minimize} \quad -\frac{1}{2}\sum_i \sum_j \alpha^i \alpha^j y^i y^j {x^i}^T x^j + \sum_i \alpha^i$$

$$\text{such that} \quad 0 \leqslant \alpha^i \leqslant C$$

$$\sum_i \alpha^i y^i = 0$$

The main takeaway here, is that the hyperplane parameters have disappeared from the minimization objective. The only parameters that remain are one alpha per instance in our data, and the hyperparameter C. The alpha's function as an encoding of the support vectors: any instance for which the corresponding alpha is not zero is a support vector.

The second thing to notice is that the algorithm only operates on the **dot products** of pairs of instances. In other words, if you didn't have access to the data, but I did give you the full matrix of all dot products of all pairs of instances, you would still be able to find the optimal support vectors.

the kernel trick

If you have an algorithm which operates only on the dot products of instances, you can substitute the dot product for a kernel function.

Which brings us to the **kernel trick**. What if I didn't give you the actual dot products, but instead gave you a different matrix of values, that behaved like a matrix of dot products.

a kernel function

$$k(\mathbf{x}^i, \mathbf{x}^j)$$

A function that computes the dot product of x^i and x^j in a high-dimensional feature space, without explicitly computing those features.

There are many functions that compute the dot product of two vectors in a highly expand feature space, but don't actually require you to expand the features.

$$\text{minimize} \quad -\frac{1}{2}\sum_i \sum_j \alpha^i \alpha^j y^i y^j k(\mathbf{x}^i, \mathbf{x}^j) + \sum_i \alpha^i$$

$$\text{such that} \quad 0 \leqslant \alpha^i \leqslant C$$

$$\sum_i \alpha^i y^i = 0$$

For these functions, we can substitute the dot product with the kernel function and compute a hyperplane in this high-dimensional feature space.

Note that this only works because we rewrote the optimization objective to get rid of **w** and b. Since **w** and b have the same dimensionality as the features.

---

extending the feature space

x_1	x_2	x_1^2	x_1x_2	x_2^2	
3	105	9	315	11025	male
1	110	1	110	12100	male
7	119	49	833	14161	male
8	120	64	960	14400	male
9	120	81	1080	14400	male
12	119	144	1428	14161	female
8	122	64	976	14884	female
8	125	64	1000	15625	female
9	125	81	1125	15625	female
9	132	81	1188	17424	male
14	128	196	1792	16384	female

Let's look at an example. The simplest way we saw to extend the feature space was to add **all cross-products**. Let's have a look at a kernel that can do this for us.

$$\mathbf{a} = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}, \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

$$k(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T \mathbf{b})^2$$

$$\mathbf{a'}^T \mathbf{b'} = (\mathbf{a}^T \mathbf{b})^2$$

$$\mathbf{a'} = \begin{pmatrix} {a_1}^2 \\ {a_2}^2 \\ \sqrt{2} a_1 a_2 \end{pmatrix}, \mathbf{b'} = \begin{pmatrix} {b_1}^2 \\ {b_2}^2 \\ \sqrt{2} b_1 b_2 \end{pmatrix}$$

Here are two 2D feature vectors. What if, instead of computing their dot product, we computed the square of their dot product. It turns out that this is equal to the dot product of two 3D vectors **a'** and **b'**.

polynomial kernel

$$k(a, b) = (a^T b + 1)^d$$

feature space for d=2: all squares, all cross products, all single features

feature space for d=3: all cubes and squares, all two-way and three-way cross products, all single features.

This is a **big** feature space.

The polynomial kernel extends this idea. By adding a + 1, we don't lose the original features, and by increasing the exponent, we can add higher cross-products. For high d, we very quickly get a huge feature space. Yet all we have to do is compute one dot product,

d is a hyperparameter: increasing it does not make the algorithm much more expensive, but it increases you feature space so much that you seriously risk overfitting.

RBF kernel

$$k(a, b) = \exp(-\gamma ||a - b||)$$

feature space: infinite dimensional

Gamma is another hyperparameter.

kernels in data space

text, **DNA**, **proteins**: String kernels (inspired by edit distance)

graphs: Weisfeiler Lehman-distance

One of the most interesting application areas of kernel methods is places where you can turn a distance metric in your data space directly into a kernel, without first extracting features.

For instance for an email classifier, you don't need to extract word frequencies, ads we've done so far, you can just design a kernel that operates directly on strings (usually baed on the edit distance). If you're classifying graphs, you can design a kernel based on the Weisfeiler-Lehman graph distance metric.

using kernel SVMs

Normalize your data.

Pick a kernel (linear, RBF, polynomial)

Pick a C and hyperparams for your kernel (d, γ)

```
In [106]: from sklearn.svm import SVC

          lin = SVC(kernel='rbf', gamma=0.1, C=10)
          lin.fit(x, y)
```

76

why did neural networks come back?

Quadratic vs. linear training time.

SVM training needs to see all pairs of instances: $O(N^2)$
Neural net training needs k passes over the data: $O(kN)$

Good SVM performance required hand-designed kernels.

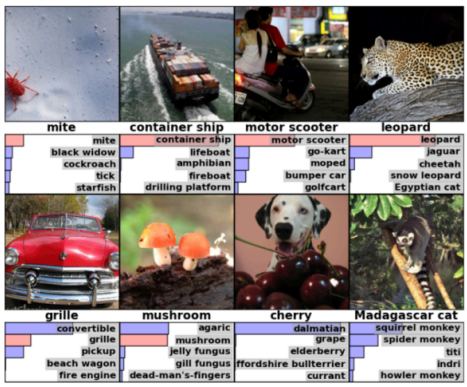
Deep neural nets matured, and hardware caught up with them.

LSTMs and ConvNets, both invented in 1998, provided the first breakthroughs.

Machine learning culture changed: empirical evidence of performance became acceptable without proof of convergence/learnability.

77

Neural nets require a lot of passes over the data, so it takes a big dataset before kN becomes smaller than N_2, but eventually, we got there. At that point, it became more efficient to train models by gradient descent, and the kernel trick lost its luster.



78

And when neural networks did come back, they caused a revolution. That's where we'll pick things up next lecture.

mlcourse@peterbloem.nl